On computational complexity of graph inference from counting

Szilárd Zsolt Fazekas, Hiro Ito, Yasushi Okuno, Shinnosuke Seki, and Kei Taneishi

September 25, 2012

Abstract

In *de novo* drug design, chemical compounds are quantitized as real-valued vectors called chemical descriptors, and an optimization algorithm runs on known drug-like chemical compounds in a database and outputs an optimal chemical descriptor. Since structural information is needed for chemical synthesis, we must infer chemical graphs from the obtained descriptor. This is formalized as a graph inference problem from a real-value vector. By generalizing subword history, which was originally introduced in formal language theory to extract numerical information of words and languages based on counting, we propose a comprehensive framework to investigate the computational complexity of chemical graph inference. We also propose a (pseudo-)polynomial-time algorithm for inferring graphs in a class of practical importance from spectrums.

1 Introduction

Structure elucidation determines the structure of chemical compounds from the information provided by various analyses such as spectroscopy, mass spectrometry, and elemental analysis. Synthetic chemistry attaches great importance to this process since it is not until its structure is determined that a chemical

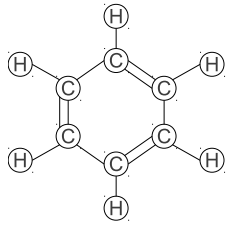

Figure 1: A chemical graph of benzene.

compound is considered synthesizable. In a slightly-different context, structure elucidation appears as an essential step in *de novo* drug design based on kernel methods [2, 3]. In a novel kernel-method-based drug design, each chemical compound in the object space is mapped to a vector of reals in the *feature (vector) space* (also called *chemical descriptor space*) according to some function f, and an optimization algorithm runs in the feature space to find an optimal vector. At this point, one has to retrieve a chemical compound that is mapped to the optimal vector by f, or favorably, enumerate the candidates of such compounds. For this purpose, stochastic and heuristic methods [2, 3] as well as polynomial-time algorithms [1, 7, 15] have been proposed.

These methods model chemical compounds as a (directed or undirected) graph each of whose vertices is labeled with a letter in an alphabet Σ, which specifies the kind of atom represented by the vertex. Such a graph is called a *chemical graph* (a.k.a., *molecular graph*). As an example, a chemical graph of benzene is illustrated in Figure 1. It is the chemical graphs of compounds that are quantified in these methods into a vector in the feature space. One significant quantification is *counting*. In fact, the above-mentioned algorithms [1, 7, 15] as well as the *spectrum kernel method* [11] studied the graph inference problem based on counting; after the spectrum kernel method we call the vector obtained by counting a *spectrum* in this paper. On the benzene graph, let us count the number of vertices of label `C` (carbon), that of `CH`-walks (edges between a carbon and a hydrogen) as well as that of `CCH`-walks, and we obtain 6, 6, 18, respectively. These counts amount to a spectrum. The problem of our interest is that given a vector like $(6, 6, 18)$, one is required to decide whether there is a chemical graph that contains exactly 6 carbons, 6 `CH`-walks, and 18 `CCH`-walks. Enumerating all such candidates is more desirable. (With the input $(6, 6, 18)$, the answer to the decision problem is yes due to benzene, whereas for the enumeration, benzene is only one of possible answers.)

This paper has a two-fold purpose. The first is to propose a comprehensive framework for the above-mentioned graph inference *decision* problem from a given vector obtained by counting. Let us give the name Solvability to this framework (its formal definition will be given in Section 3). Solvability is a class of decision problems, each of whose elements is identified by:

1. the class of counting-based functions used to compute a feature vector; and

2. the class of inferred graphs.

By saying "counting-based" in the first criterion, we mean that our interests lie not only in the functions that simply count vertices or walks but also in functions that apply arithmetic operations on these counts. Consider functions g_1 and g_2 that count the numbers of `C` and `CH`, respectively, and let $h = g_1 - g_2$. We regard h as a counting-based function. In the framework Solvability, inferring a graph whose h-value is 0 means inferring a graph that contains the same number of carbons as `CH`-walks, such as benzene.

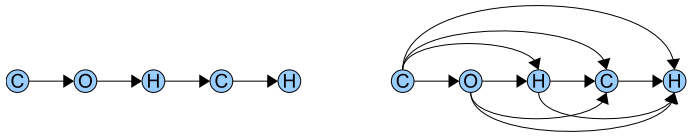

Figure 2: (Left) The continuous subword graph of a word `COHCH`. (Right) The scattered subword graph of the same word.

As for the second criterion, we are required to find classes of graphs that are descriptive enough for applications such as drug design and simple enough to make the class in Solvability thus identified efficiently solvable. The most concise graph for which inference from counts is meaningful is a "word." A word α, i.e., a sequence of letters, can be naturally modeled as a directed graph each of whose nodes corresponds to its letters and is connected towards its successor by a directed edge (see Figure 2). Not only counting letters (*Parikh mapping* [16]) but also counting continuous subwords have been widely applied, e.g., n-gram in indexing for full text search [20]. A word u is a *continuous subword* of α if $\alpha = xuy$ for some words x, y. Counting occurrences of u as a continuous subword of α is equivalent to counting (directed) u-walks on the corresponding graph, and due to this property, we call the graph the *continuous subword graph* of α. Surprisingly, Solvability was recently proved undecidable even on such simplest objects, as long as we are allowed to employ as a mapping function those obtained by recursively composing simple counting functions by pointwise addition, pointwise multiplication, and multiplication by -1 [19]. Intuitively, this is because the function class is complex enough to let Solvability "solve" Diophantine equations, and it remains so even when $|\Sigma| \leq 9$. Needless to say, words, or their sequential graph models, are too simple to describe chemical compounds, for which cyclic structures such as a benzene ring are indispensable. This means that in order for a class in Solvability to be solvable in a practical amount of time with sufficiently descriptive graph class, the class of (counting-based) functions it considers should be more restricted than the one just mentioned.

The second purpose of this paper should be clear now; characterizing pairs of a class of counting-based functions and a class of graphs to be inferred for which Solvability can be solved in a polynomial time and is useful in drug design. As such a pair, we consider the simple counting and the class of graphs of *tree-width 2*. Tree-width of a graph measures how similar the graph is to the tree, and will be explained in detail in Section 4.2; note that a graph of tree-width 1 is a tree and vice versa. The class of graphs of tree-width 2 has practical significance as illustrated in database search (see Table 1).

A brief outline of this paper follows. After giving preliminaries in Section 2, in Section 3, we give a definition of walk history with some existing as well as new results on the computational complexity of Solvability. In Section 4, we propose a (pseudo) polynomial-time algorithm for Solvability with the object

Tree-width	# compounds	
1	1881	19.4%
2	7336	75.7%
3	477	4.9%
4	1	0.01%
≥ 5	0	
total	9695	

Table 1: Chemical compounds in LIGAND database [9] are analyzed on their tree-width [21].

space being the set of graphs of tree-width at most 2 and the function being spectrum extraction.

2 Preliminaries

In this section, we introduce necessary notions and notation from graph theory and formal language theory. For more details, see [5, 18].

Let Σ be an alphabet. A word α is a sequence of letters $a_1, a_2, \ldots, a_n \in \Sigma$ for some $n \geq 0$, that is, $\alpha = a_1 a_2 \cdots a_n$. This n is called the *length* of α, and it is denoted by $|\alpha|$. If $n = 0$, then this word is specially called the *empty word*, which is denoted by λ. Let Σ^* be the set of all words over Σ and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. By $\Sigma^{\leq k}$, we denote the set of all words over Σ of length at most k. We employ an alphabet $\Sigma_a = \{\mathtt{C}, \mathtt{O}, \mathtt{H}\}$ for examples in this paper, where `C`, `O`, and `H` stand for carbon, oxygen, and hydrogen, respectively. For instance, ${\Sigma_a}^{\leq 2} = \{\lambda, \mathtt{C}, \mathtt{O}, \mathtt{H}, \mathtt{CC}, \mathtt{CO}, \mathtt{CH}, \mathtt{OC}, \mathtt{OO}, \mathtt{OH}, \mathtt{HC}, \mathtt{HO}, \mathtt{HH}\}$.

2.1 Multigraphs

A (Σ-*labeled*, loopless, undirected) *multigraph* is a triple $G = (V, E, \ell)$ of disjoint sets V, E and a labeling function $\ell : V \to \Sigma$ (V is also denoted by $V(G)$). Each vertex is labeled with a letter in Σ, and a vertex v with the label $a \in \Sigma$, that is, $\ell(v) = a$, is called an *a-vertex (according to ℓ)*. An element $e \in E$ is mapped to a set of two *distinct* vertices $v_1, v_2 \in V$, and called an edge between v_1 and v_2 (the distinctness of v_1 and v_2 is required for G to be loopless). Such an edge e is written as $e = v_1 v_2$. We write $v \in G$ and $v_1 v_2 \in G$ for $v \in V$ and $v_1 v_2 \in E$, respectively. The *degree* (a.k.a., *valency*) of a vertex $v \in V$ is the number of edges that contain v, and it is denoted by $d(v)$. The *degree* of G is defined to be $\max\{d(v) \mid v \in V\}$. For a subset V' of V, the subgraph of G that is composed of all the vertices in V' and all the edges $v_1 v_2 \in E$ with $v_1, v_2 \in V'$ is said to be *induced by V'*, and denoted by $G[V']$. There may be more than one edge between two vertices in a multigraph; if a multigraph has the property that

there exists no more than one edge between any of its two vertices, then the multigraph is rather called a *simple graph*.

A *walk* of length l in G is an alternating sequence $\pi = v_0e_0v_1e_1\cdots e_{l-1}v_l$ of vertices and edges in G such that $e_i = v_iv_{i+1}$ for $0 \leq i < l$. Note that on a walk a vertex may appear more than once. When the v_i's are all distinct, we call this walk a *path*. In order to label walks, ℓ is extended as $\ell(\pi) = \ell(v_0)\ell(v_1)\cdots\ell(v_l)$. A walk with the label $u \in \Sigma^*$ is called a *u-walk*. A multigraph is said to be *connected* if between any two of its vertices there is a path.

By $\mathcal{G}$, we denote the class of Σ-labeled, loopless, connected, undirected multigraphs. For any graph class $\mathcal{G}'$ and an integer $\Delta \geq 1$, we write $\mathcal{G}'(\Delta)$ for the class of graphs in $\mathcal{G}'$ whose maximum degree is at most Δ; this Δ is called a *degree bound*.

Given an undirected graph $G = (V, E)$, orienting each edge in E transforms G into a *directed* graph $\overrightarrow{G}$. An edge v_1v_2 in G is considered to be either directed from v_1 to v_2 or directed oppositely in $\overrightarrow{G}$. In order to specify the direction clearly, we write $\overrightarrow{v_1v_2}$ and mean that this edge is oriented from v_1 to v_2. On a directed graph, a walk $\pi = v_0e_0v_1e_1\cdots e_{\ell-1}v_\ell$ is required to satisfy the extra condition that $e_i = \overrightarrow{v_iv_{i+1}}$ for any $0 \leq i < \ell$. A directed graph is *weakly-connected* if replacing all of its edges with undirected edges produces a connected graph. When we say a directed graph is connected below, we mean that the graph is weakly-connected (there is another criterion of connectivity called strong connectivity, which requires the existence of a walk between any two vertices, but this is not the connectivity criterion we consider in this paper).

Let us denote by $\overrightarrow{\mathcal{G}}$ the directed analog of $\mathcal{G}$. We denote any class of directed graphs with the over-right-arrow. Any undirected graph can be considered directed by replacing each edge v_iv_j by two directed edges $\overrightarrow{v_iv_j}$ and $\overleftarrow{v_iv_j}$. Thus, we can say that $\mathcal{G}$ is a subclass of $\overrightarrow{\mathcal{G}}$.

2.2 Trees and series-parallel graphs

Two important subclasses of $\mathcal{G}$ are the classes of trees and of series-parallel graphs. A path of length at least 2 between two vertices and an edge between them form a *cycle*. A (*rooted*) *tree* is a cycle-free simple graph, in which one vertex is regarded as its *root* for convenience. The root endows a tree with parent-child relation among its vertices, and this relation, in turn, enables us to define the notions of subtree and height. Let T be a tree. The *subtree* of T at $t \in V(T)$, which is denoted by $\mathrm{sub}(T, t)$, is the subgraph induced by the node t and all its descendants (not only the direct ones) with respect to the parent-child relation. Note that, between any vertices $t_1, t_2 \in T$, there is a (unique) path, which is written as t_1Tt_2. The vertices at distance h from the root have *height* h, and form the h-th level of T. The largest height in T is called the *height* of T, and it is denoted by $h(T)$. We denote the class of trees by Υ, and the class of (rooted) trees of height at most h by Υ_h. It is important that, for any fixed degree bound $\Delta \geq 1$ and a height $K \geq 0$, the size of $\Upsilon_K(\Delta)$ is constant, whereas that of Υ_K can be arbitrary large.

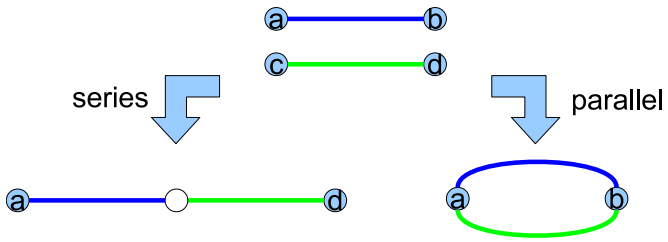

Figure 3: Series and parallel operations. Series operation merges the vertex v_b (only the subscript is written in this figure for clarity) with the vertex v_c and produces the left graph with terminals v_a and v_d. The merged vertex is indicated by the white circle. Parallel operation merges v_a with v_c and v_b with v_d, respectively, and results in the right graph with v_a, v_b being terminals.

A *series-parallel graph* is a graph with two special vertices called *terminals*, and defined recursively by applying series operation and parallel operation (see Figure 3). The graph that consists of two vertices v_a and v_b connected by an edge is a series-parallel graph whose terminals are v_a and v_b. Let us say that this is *atomic* or call this an *atom*. When two series-parallel graphs with respective terminals v_a, v_b and v_c, v_d are given, merging v_b with v_c and designating v_a, v_d terminals is called the *series operation*, and the graph thus obtained is also defined to be series-parallel. Merging v_a with v_c and v_b with v_d is called the *parallel operation*, and the resulting graph is also defined to be series-parallel. Any vertices but terminals are *internal*. We denote the class of series-parallel graphs by $\mathcal{SPG}$. Series-parallel graphs are, by definition, planar, and hence, $\mathcal{SPG}$ is a subset of the class $\mathcal{PLG}$ of planar graphs.

3 Walk history in graphs

Given a directed graph $\overrightarrow{G}$, we can count the number of u-walks in it, which we denote by $|\overrightarrow{G}|_u$. Throughout this paper, we make great use of the notations of the form $|A|_B$, an object A sandwiched by vertical bars accompanied by a set B of objects as a subscript, in order to represent the non-negative integer vector of the number of occurrences of each object included in B on A ordered in some standardized manner. According to this criterion, $|\overrightarrow{G}|_u$ should have been written rather as $|\overrightarrow{G}|_{\{u\}}$, but we use the former whenever the set B is singleton. In the introduction, we mentioned informally that counting labeled walks on a graph is a generalization of counting subwords on a word. Let us explain this more formally first, and then propose a framework Solvability for the graph inference problem from a vector obtained by counting-based functions.

Let $\alpha = a_1 a_2 \dots a_n \in \Sigma^*$ be a word on which we will count subwords. A word u is a *scattered subword* of α if there are an integer $k \geq 1$ and words

$x_1, \ldots, x_k, y_0, y_1, \ldots, y_k \in \Sigma^*$ such that $u = x_1 \cdots x_k$ and $\alpha = y_0 x_1 y_1 \cdots x_k y_k$ (if $k = 1$, then u is especially called a *continuous subword* of α). The *scattered subword graph* of α, which we denote by $SSG(\alpha)$, is a directed simple graph $(\{v_1, v_2, \ldots, v_n\}, E, \ell)$ with $E = \{\overrightarrow{v_i v_j} \mid i < j\}$ and $\ell(v_i) = a_i$ for $1 \leq i \leq n$. Its *continuous subword graph*, denoted by $CSG(\alpha)$, is a directed simple graph $(\{v_1, v_2, \ldots, v_n\}, E, \ell)$, where $E = \{\overrightarrow{v_i v_{i+1}} \mid 1 \leq i < n\}$ and $\ell(v_i) = a_i$ for $1 \leq i \leq n$. See Figure 2 for an example of these graphs. It is obvious that $|SSG(\alpha)|_u$ and $|CSG(\alpha)|_u$ are equal to the number of occurrences of u as a scattered subword of α and to the number of occurrences as its continuous subword, respectively. By $\overrightarrow{\mathcal{SSG}}$ ($\overrightarrow{\mathcal{CSG}}$), we denote the class of scattered (resp. continuous) subword graphs.

Now we generalize the notion of subword history proposed by Mateescu, Salomaa, and Yu [12] onto the class of directed multigraphs $\overrightarrow{\mathcal{G}}$ in the name of walk history.

Definition 1. A *walk history* in Σ and its *value* in a graph $G \in \overrightarrow{\mathcal{G}}$ are defined recursively as follows:

- Every u in Σ^* is a walk history, referred to as *monomial*, and its value in G equals $|G|_u$. Hence, λ is also a monomial, and we assume that its value is always 1.
- Assume that WH_1 and WH_2 are walk histories with values n_1 and n_2 in G, respectively. Then $-(WH_1)$, $(WH_1)+(WH_2)$, and $(WH_1)\times(WH_2)$ are walk histories with respective values $-n_1$, $n_1 + n_2$, and $n_1 \times n_2$ in G.

A monomial u can be considered as a function from $\overrightarrow{\mathcal{G}}$ to the set of nonnegative integers $\mathbb{N}_0$ that outputs $|G|_u$, the number of u-walks on a given $G \in \overrightarrow{\mathcal{G}}$. From this viewpoint, u is merely a simple counting function. A walk history is a composition of such simple counting functions by pairwise addition, pairwise multiplication, and multiplication by -1. A walk history is *linear* if it can be obtained from monomials without using the operation $\times$. By this definition, a monomial is a linear walk history. For a walk history WH, we denote the value of WH in a graph G by $|G|_{WH}$. By restricting the graph class to that of scattered subword graphs $\overrightarrow{\mathcal{SSG}}$, the above definition coincides with that of subword history proposed in [12].

A *system S of walk histories* is an ordered set of walk histories $WH_1, \ldots, WH_m$ for some $m \geq 1$. For a graph G, we denote the vector $(|G|_{WH_1}, \ldots, |G|_{WH_m})$ by $|G|_S$. We call a system of monomial walk histories a *counting system*. The classes of systems of walk histories, of systems of linear walk histories, and of counting systems are denoted by $\mathcal{SWH}$, $\mathcal{SLWH}$, and $\mathcal{COUNT}$, respectively, where the first letter $\mathcal{S}$ stands for system. By $\mathcal{WH}$ ($\mathcal{LWH}$), we denote the class of systems of single (linear) walk history. These notations are parameterized by a nonnegative integer like $\mathcal{WH}(1)$ to devise a notation for their subclass where the length of monomials in walk histories involved is bounded by the parameter. For example, $a + b \in \mathcal{WH}(1)$ but $a + bc \notin \mathcal{WH}(1)$.

A subclass of $\mathcal{COUNT}(K+1)$ has commanded special attention. That is the class consisting of only one element, which is a system of all monomials of length at most $K+1$ ordered in some standardized manner. The notation $\boldsymbol{S}_K$ is used for this element and for this class interchangeably. For a graph G, $|G|_{\boldsymbol{S}_K}$ is often called the *feature vector of G of level K* [1, 15]. This vector is obtained by counting the number of u-walks for every $u \in \Sigma^{\leq K+1}$ and ordering them in the standardized manner. Given a feature vector $\boldsymbol{v}$ of level K and $u \in \Sigma^{\leq K+1}$, we refer to the component of $\boldsymbol{v}$ that corresponds to u by $\boldsymbol{v}(u)$.

Given a system S of walk histories and an integer vector $\boldsymbol{v}$, we can consider the problem of deciding whether there exists a graph G such that $|G|_S = \boldsymbol{v}$. A class $\mathcal{C}_1$ of systems of walk histories and a class $\mathcal{C}_2$ of graphs define the following decision problem:

$$\textsc{Solvability}(\mathcal{C}_1, \mathcal{C}_2) = \left\{ \langle S, \boldsymbol{v} \rangle \;\middle|\; \begin{array}{l} S \in \mathcal{C}_1,\ \boldsymbol{v} \in \mathbb{N}_0^{|S|}, \text{ and there exists} \\ \text{a graph } G \in \mathcal{C}_2 \text{ such that } |G|_S = \boldsymbol{v}. \end{array} \right\}$$

For example, $\textsc{Solvability}(\boldsymbol{S}_0, \overrightarrow{\mathcal{CSG}})$ is the problem of finding a continuous subword graph that contains exactly the same number of a-vertices as specified by a given input for every letter $a \in \Sigma$. Thus, this can be interpreted as the problem of finding a word whose Parikh mapping is equal to a vector given as input (this problem is though trivial since such a word is always found). Based on Matiyasevich's results on Hilbert's 10th problem [13, 14], Seki proved the following result:

Theorem 1 ([19])**.** $\textsc{Solvability}(\mathcal{WH}(1), \overrightarrow{\mathcal{SSG}})$ *is undecidable even over 9-letters alphabet.*

Remark 1. In fact, in Theorem 1, $\overrightarrow{\mathcal{SSG}}$ can be replaced with $\overrightarrow{\mathcal{CSG}}$ without destroying the undecidability. This is because as long as only the monomials of length at most 1 are concerned, there is no difference between $\overrightarrow{\mathcal{SSG}}$ and $\overrightarrow{\mathcal{CSG}}$. However, this is not so essential.

Corollary 2 in [12] states that for a walk history WH, there exists a linear walk history LWH such that for any $G \in \overrightarrow{\mathcal{SSG}}$, $|G|_{WH} = |G|_{LWH}$. Thus, $\textsc{Solvability}(\mathcal{LWH}, \overrightarrow{\mathcal{SSG}})$ is undecidable for a 9-letter alphabet. This linearization may involve increasing the length of monomials. Actually, once the length of monomials is restricted to be 1, this undecidability does not hold any more. In fact, $\textsc{Solvability}(\mathcal{LWH}(1), \overrightarrow{\mathcal{SSG}})$ is equivalent to $\textsc{Solvability}(\mathcal{LWH}(1), \overrightarrow{\mathcal{CSG}})$ due to the reason mentioned in Remark 1, and it is proved decidable as a corollary of the following theorem for the class of systems of linear walk histories $\mathcal{SLWH}$.

Theorem 2. $\textsc{Solvability}(\mathcal{SLWH}, \overrightarrow{\mathcal{CSG}})$ *is decidable.*

Proof. Let us begin this proof by recalling that there is a 1-to-1 correspondence between the continuous subword graphs in $\overrightarrow{\mathcal{CSG}}$ and the words in Σ^* and that counting u-walks on a graph in $\overrightarrow{\mathcal{CSG}}$ is equivalent to counting occurrences of u as a *continuous* subword on the corresponding word.

In Solvability($\mathcal{SLWH}, \overrightarrow{\mathcal{CSG}}$), the following pair is given as input: a system of linear walk histories $S = \{WH_1, \ldots, WH_k\}$ and a k-dimensional vector $\boldsymbol{v} = (n_1, \ldots, n_k)$ for some $k \geq 0$. Being linear, WH_i can be written as $c_1u_1 + \cdots + c_mu_m$ for some $m \geq 0$ and integer constants $c_1, \ldots, c_m$. Let us construct a finite automaton augmented with two 1-reversal counters that computes $|CSG(\alpha)|_{WH_i}$ for a given word α (counters are a unary stack, and 1-reversal counters are a counter that cannot increment once being decremented; see [10] for details). Its input head always performs look-ahead by $\max\{|u_1|, \ldots, |u_m|\}$, and once it finds u_j, it increments the first (second) counter by $|c_j|$ if c_j is positive (resp. negative). After the head scans the input, this machine subtracts the value of the second counter from that of first, which is $|CSG(\alpha)|_{WH_i}$. In this way, we can design a finite automaton M with $2k$ 1-reversal counter machines that computes $|CSG(\alpha)|_S$, and accepts α if and only if this value is equal to $\boldsymbol{v}$. Thus, M accepts some word if and only if there exists a graph G in $\overrightarrow{\mathcal{CSG}}$ such that $|G|_S = \boldsymbol{v}$. Now it suffices to mention the known result that emptiness test of 1-reversal counter machines is decidable [10]. □

3.1 Solvability on trees and tree-like structures

For arbitrary but fixed K and Δ, Akutsu and Fukagawa [1] have proposed an algorithm for Solvability($\boldsymbol{S}_K, \Upsilon(\Delta)$), that is, the problem of deciding whether there is a tree of degree at most Δ which contains the number of u-walks as specified by an input vector for every $u \in \Sigma^{\leq K+1}$. We will call this algorithm *A-F algorithm* in this paper and give a brief explanation of how it works in Section 4.1. Its running time is polynomial in the size of output trees, and actually it works in pseudo-polynomial time. On the other hand, Solvability($\boldsymbol{S}_4, \mathcal{TW}(2)$) is strongly NP-hard [1], where $\mathcal{TW}(2)$ is the class of graphs of tree-width at most 2 (the definition of treewidth can be found in Section 4.2). A related but incomparable NP-hardness result holds as follows, whose proof is found in Section 6.1 (note that $\mathcal{TW}(2)$ is a proper subclass of the class $\mathcal{PLG}$ of planar graphs).

Theorem 3. Solvability($\boldsymbol{S}_2, \mathcal{PLG}$) *is strongly* NP*-hard.*

This 2 is the lower bound in this context due to a pseudo-polynomial-time algorithm proposed by Nagamochi [15] for Solvability($\boldsymbol{S}_1, \mathcal{G}$). For the strong NP-hardness and pseudo-polynomial-time algorithm, see [8].

4 Graph inference based on tree-decomposition

In this section, we propose an algorithm to infer from $\boldsymbol{S}_K$ graphs of tree-width at most w with bounded degree Δ, where $w, \Delta \geq 1$ and $K \geq 0$ are constants given *a priori*. Then, we analyze a case, which is significant in practice, when our algorithm works in pseudo-polynomial time. That is when $w = 2$ (see Table 1).

One thing to note is that though we have been working on the class $\overrightarrow{\mathcal{G}}$ of directed graphs, in this section we focus our attention on the class $\mathcal{G}$ of undirected graphs. This is not because our algorithm does not work on directed graphs;

actually it does. This is to prevent taking edge directions into account from making our explanation less understandable. Another reason is that chemical compounds are usually modeled as undirected graphs.

4.1 Akutsu and Fukagawa's algorithm (A-F algorithm)

The algorithm we propose is a modification of A-F algorithm, which infers from $\boldsymbol{S}_K$ trees with bounded degree Δ. For this reason, its review should be helpful. A-F algorithm takes a $|\Sigma^{\leq K+1}|$-dimension non-negative integer vector $\boldsymbol{v}$ as an input, and decides whether there is a (Σ-labeled) tree $T \in \Upsilon(\Delta)$ whose feature vector of level K is equal to $\boldsymbol{v}$. Note that the number of vertices of such T, if any, should be $\sum_{a\in\Sigma} \boldsymbol{v}(a)$. Let us denote the sum by n below.

There is a notion concerning to the tree that is essential for A-F algorithm. That is the frontier vector. Recall that, given a (rooted) tree $T \in \Upsilon(\Delta)$, whose degree is bounded by Δ, its vertices at distance h from its root form the h-th level of T. We collectively call the $h(T), h(T)-1, \ldots, h(T)-K$ levels of T its *frontier of level* K, and denote it by ∂T_K. For instance, ∂T_0 contains only the root of T. The T's frontier of level K is actually a forest, and all of trees contained are in $\Upsilon_K(\Delta)$ because the degree of T is bounded by Δ. As the size of $\Upsilon_K(\Delta)$ is constant, searching the frontier for each tree in $\Upsilon_K(\Delta)$ and counting its occurrences produces a constant-dimension vector called the *T's frontier vector of level K*. We denote this vector by $|\partial T_K|$.

A-F algorithm uses dynamic programming to fill a Boolean-valued table D whose cells are referred to by a feature vector and a frontier vector (of a tree). All the D's cells are initialized `false`, and this algorithm updates cells in the way described shortly such that $D(\boldsymbol{v}', \boldsymbol{f}') =$ `true` if and only if there exists a tree T_1 of degree at most Δ whose feature vector of level K is $\boldsymbol{v}'$ and whose frontier vector of level K is $\boldsymbol{f}'$. Once being built, this table is referred to by A-F algorithm in order to decide the problem; the answer is yes if and only if $D(\boldsymbol{v}, \boldsymbol{f}) =$ `true` for some $\boldsymbol{f}$. It may be noteworthy that the sum of the values of all components of such $\boldsymbol{f}$ is at most $(K+1)n$.

Let us explain how A-F algorithm updates D. This proceeds in a dynamic programming manner based on the fact that, starting from a tree of one vertex, any tree can be constructed by iteratively appending a new vertex to a tree at its deepest or second deepest level. (Given a tree, one can index its vertices in the breadth-first-search, and these numbers specify the order of adding vertices to build the tree in the above-mentioned manner.) First of all, for each letter $a \in \Sigma$, the tree T_a of only one a-vertex has this algorithm update the cell $D(|T_a|_{\boldsymbol{S}_K}, |\partial T_{aK}|)$ to be `true`. After this initialization is done, A-F algorithm recursively updates other cells according to the cells that have been already updated to be `true` like $D(|T_a|_{\boldsymbol{S}_K}, |\partial T_{aK}|)$ just mentioned above. Let $D(\boldsymbol{v}', \boldsymbol{f}')$ be such a cell referred to by a feature vector $\boldsymbol{v}'$ and a frontier vector $\boldsymbol{f}'$ of level K. By definition, there must exist a tree T_1 whose feature vector and frontier vector of level K are $\boldsymbol{v}'$ and $\boldsymbol{f}'$, respectively. Consider a process to add a new vertex to this tree T_1 in the way mentioned above so as to generate a tree T_2, and see how feature and frontier vectors of T_2 are computed from those of T_1.

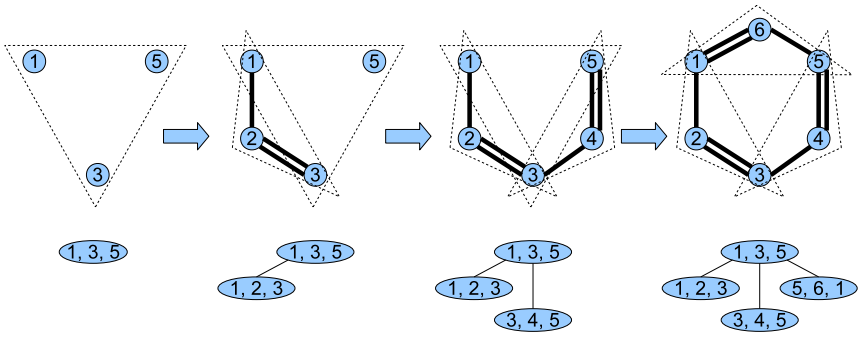

Figure 4: A growth of benzene graph, and the corresponding growth of a decomposition tree.

For each $u \in \Sigma^{\leq K+1}$, $|T_2|_u$ is equal to $|T_1|_u$ plus the number of u-walks just created by this addition. On these new u-walks lies no vertex that was outside of the T_1's frontier of level K as they are too far from the added vertex. Thus, this update needs only the T_1's frontier of level K. The number of these new walks is completely determined by the label of added vertex and to which tree (in $\Upsilon_K(\Delta)$) in the T_1's frontier of level K the vertex was added. Let $\boldsymbol{v}''$ and $\boldsymbol{f}''$ be the T_2's feature and frontier vectors of level K, respectively. Then the cell $D(\boldsymbol{v}'', \boldsymbol{f}'')$ is updated to be `true`. In this way, the D's cells are updated recursively one after another.

Since the size of Σ and that of $\Upsilon_K(\Delta)$ are bounded, one `true` cell of D thus updates a constant number of other cells to be `true` (we have not explained how the frontier vector of T_2 is computed, but it should be now almost straightforward, and hence, is omitted). How many cells does D need in total? With its usage by A-F algorithm and this updating method in mind, D needs only the cells that are referred to by a feature vector smaller than the input $\boldsymbol{v}$ according to pairwise component comparison and by a frontier vector the sum of whose components does not exceed $(K+1)n$. Thus, the number of cells in D is $O(n)$. Therefore, the construction of D can be done in polynomial time in n, and this process dominates the time complexity of A-F algorithm.

4.2 An outline of proposed algorithm and tree decomposition

The above-mentioned idea of A-F algorithm should be general enough to be applied to tree-like structures. We focus attention on the *tree-decomposition* of a graph, proposed by Robertson and Seymour [17]. A graph can be characterized by a positive integer called *tree-width* (the notions of tree-decomposition and tree-width will be formally introduced shortly, and here we only note that the

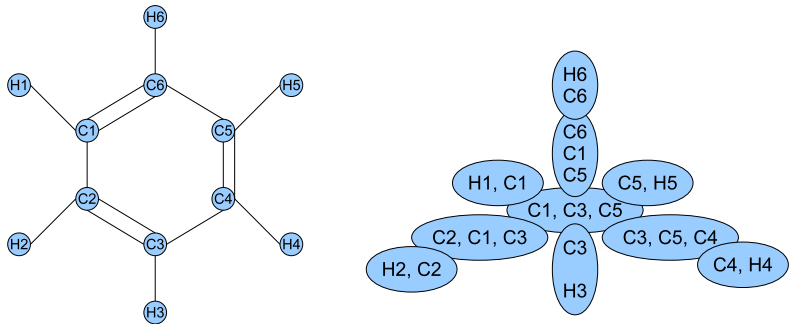

Figure 5: A chemical graph G_b of benzene and the decomposition tree T_b of G_b.

tree-width of trees is 1). A graph G of tree-width $w \geq 1$ decomposes into a tree of subsets of $V(G)$ of size at most $w+1$ (*decomposition tree of width at most* w). We will redesign A-F algorithm so as to run dynamic programming based on the growth of decomposition trees of width at most w (see Figure 4 for an example of this growth) instead of the growth of trees.

In order to introduce the notion of tree-decomposition formally, for a graph G and a tree T, let us consider a way to assign each tree vertex $t \in T$ with a nonempty subset of $V(G)$, which is denoted by V_t. Let $\mathcal{V} = \{V_t \mid t \in T\}$, and we call its elements *bags* (in the original definition of tree-decomposition [5], the non-emptiness of a bag is not assumed, but the conditions (T2) and (T3), introduced below, justify this assumption). The pair $(T, \mathcal{V})$ is called a *tree-decomposition* of G if the next three conditions hold:

(T1) for every vertex $v \in G$, there is a tree vertex $t \in T$ such that $v \in V_t$;

(T2) for every edge $e = v_1v_2 \in E$, there exists $t \in T$ such that $\{v_1, v_2\} \subseteq V_t$;

(T3) for any $t_1, t_2, t_3 \in T$, if $t_2 \in t_1Tt_3$, that is, t_2 is on the path of T between t_1 and t_3, then $V_{t_1} \cap V_{t_3} \subseteq V_{t_2}$.

G is called the *underlying graph* of $(T, \mathcal{V})$.

As being announced, the tree-decomposition will be used in our algorithm in a similar way as the tree was used in A-F algorithm. As such, we rather call $(T, \mathcal{V})$ a *decomposition tree*. In addition, with the usage in mind, we should regard a bag V_t not merely as a set of vertices but as the subgraph of G induced by V_t, that is, $G[V_t]$. For convenience in explanations, let us extend the notation V_t to $V_{T'}$ for a subtree T' of T as: $V_{T'} = \bigcup_{t \in T'} V_t$.

The *(tree)-width* of $(T, \mathcal{V})$ is defined as $\max_{t \in T}\{|V_t|\} - 1$, that is, the size of largest bag(s) in $\mathcal{V}$ minus one. The *(tree)-width* of G is defined as the minimum among the widths of all its decomposition trees. By $\mathrm{tw}(G)$, we denote the (tree)-width of G. As an example, Figure 5 illustrates a chemical graph G_b of benzene and its decomposition tree T_b. The size of the largest bags in this tree is 3, and

hence, $\mathrm{tw}(G_b) \leq 2$. Since the class of graphs of tree-width 1 is equal to the class of trees [5], a graph with a cycle is of tree-width at least 2. Thus, $\mathrm{tw}(G_b) = 2$.

As Diestel mentions in [5], "*the most important feature of a tree-decomposition is that it transfers the separation properties of its tree to the graph decomposed*". This separation property is explained as follows. For an edge $t_1t_2 \in T$, let T_1, T_2 be the two components of $T - t_1t_2$ such that $t_1 \in T_1$ and $t_2 \in T_2$. Then $V_{t_1} \cap V_{t_2}$ separates V_{T_1} from V_{T_2} in G, that is, for any $v_1 \in V_{T_1}$ and $v_2 \in V_{T_2}$, every path between them contains a vertex in the intersection.

By $\mathcal{TW}(w)$, we denote the class of graphs of tree-width at most w. As mentioned above, $\mathcal{TW}(1) = \Upsilon$. $\mathcal{TW}(2)$ is the class of graphs all of whose biconnected components are series-parallel graphs [4]. Since any (connected) graph decomposes into a tree of biconnected components (called the *block tree* of the graph), a graph in $\mathcal{TW}(2)$ can be described as a tree of series-parallel graphs. For a degree bound $\Delta \geq 1$, $\mathcal{TW}(w, \Delta)$ denotes the class of graphs whose tree-width is at most w and whose degree is at most Δ.

Having formally introduced the decomposition tree, the notion of tree-width, and the related notions and notation, let us return to the explanation of the algorithm we propose. It is a generalization of A-F algorithm, but based on the growth of decomposition tree in place for tree. In order for this algorithm to run in polynomial time in n, it suffices to prove that there exist constants c_1, c_2 such that any graph with bounded degree whose tree-width is at most w admits a decomposition tree with the following three properties:

(P1) its degree is bounded by c_1;

(P2) its size is $O(n)$;

(P3) the level of frontier necessary for the table update is bounded by c_2.

In Section 4.3, we show that any graph of tree-width w admits a decomposition tree satisfying (P1) and (P2). As for (P3), we can prove its validity only for the case of tree-width being 1 and 2; more precisely, we can prove that any graph of tree-width at most 2 admits a decomposition tree with (P1)-(P3) (Section 4.5). This enables us to conclude that our algorithm works in a polynomial time in n when the graphs of tree-width at most 2 are sought for.

As a preliminary for examining (P1)-(P3), we first introduce the following two "overlapping" properties. The first one holds between adjacent bags of a decomposition tree as stated in the next lemma, which makes possible to visually understand the addition of a bag to a decomposition tree as superposing the bag onto a parent bag (see Figure 5).

Lemma 4. *Let G be a graph and $(T, \mathcal{V})$ be its decomposition tree. If there is an edge between t and t' in T, then $V_t \cap V_{t'} \neq \emptyset$.*

Proof. Suppose that $V_t \cap V_{t'} = \emptyset$. Since G is connected, there must exist a path in G from a vertex $v_1 \in V_t$ to a vertex $v_n \in V_{t'}$. Without loss of generality, we can assume that $v_2, \ldots, v_{n-1} \notin V_t \cup V_{t'}$ by choosing the path as short as possible. Due to (T2), for $1 \leq i < n$, $\{v_i, v_{i+1}\} \subseteq V_{t_i}$ for some $t_i \notin \{t, t'\}$.

If $t' \in t_1Tt$, then (T3) implies $u_1 \in V_{t_1} \cap V_t \subseteq V_{t'}$, but this contradicts the emptiness of $V_t \cap V_{t'}$. Thus, $t \in t_1Tt'$. If $t, t' \in t_iTt_{i+1}$ for some $1 \leq i < n$, then $v_{i+1} \in V_{t_i} \cap V_{t_{i+1}} \subseteq V_t, V_{t'}$, and hence, $V_t \cap V_{t'}$ would not be empty. Combining these together results in $t \in t_{n-1}Tt'$, but then $u_n \in V_{t_{n-1}} \cap V_{t'} \subseteq V_t$, and we reach the same contradiction. □

The second one holds between, so to speak, two subtrees induced by respective two vertices of underlying graph G. For a vertex $v \in G$, let $T_v = \{t \mid v \in V_t\}$. Due to (T3), T_v becomes a connected component (a subtree) of T.

Lemma 5 (see [5]). *$T_{v_1} \cap T_{v_2} \neq \emptyset$ whenever v_1v_2 is an edge of G.*

4.3 Normal forms of decomposition tree

Our algorithm is a dynamic-programming algorithm based on the growth of decomposition trees. Its primary purpose is to search for a graph with the feature vector given as input among all the graphs in $\mathcal{TW}(w, \Delta)$, and after all, decomposition trees are nothing more than an algorithmic subsidiary. Though a graph admits multiple decomposition trees, it suffices for our algorithm to grow one decomposition tree for each graph in $\mathcal{TW}(w, \Delta)$, and the choice of such a "representative" decomposition tree is up to us. The aim of this section is to enumerate properties that we can expect the representative to possess. It facilitates our algorithm design to choose a decomposition tree with these properties as representative. It goes without saying that tree-width being w is such a property.

The first property, the sub-connectedness, has been proposed by Fraigniaud and Nisse [6]. A decomposition tree $(T, \mathcal{V})$ of a graph G is *sub-connected at* $t \in T$ if, for all $t' \in \mathrm{sub}(T, t)$, $G[V_{\mathrm{sub}(T,t')}]$ is connected. Thus, given a decomposition tree, if it is sub-connected at its root, then its underlying graph is connected. A procedure `SPLIT` was developed in [6] to transform a given decomposition tree of G into another decomposition tree that is sub-connected at its root without any increase of width provided G is connected. This means that the sub-connectedness is compatible with the tree-width being w. More importantly, by taking only the sub-connected decomposition trees into account, our algorithm avoids the risk of inferring a non-connected graph.

`SPLIT` does not increase the size of T_v for any $v \in G$, whereas it may cause increase in the number of bags. The next lemma provides a method to decrease the number of bags to $|V(G)| - 1$ without loss of sub-connectedness.

Lemma 6. *Every graph G admits a sub-connected tree-decomposition of width* $\mathrm{tw}(G)$ *such that*

(N1) *none of its bags contains another.*

Proof. Let $(T, \mathcal{V})$ be a sub-connected decomposition tree of G of width $\mathrm{tw}(G)$. Among $t, t' \in V(T)$ satisfying $V_t \subseteq V_{t'}$, it suffices to consider only the adjacent ones. This is because by (T3), if $V_{t_3} \subseteq V_{t_1}$ for some $t_1, t_3 \in T$ and $t_2 \in t_1Tt_3$, then $V_{t_1} \cap V_{t_3} = V_{t_3} \subseteq V_{t_2}$. We delete t from T and connect t' with all vertices

that were adjacent to t on T. Furthermore, we set t' to be the root of resulting tree T' if the deleted t was the root of T. A pair $(T', \mathcal{V} \setminus \{V_t\})$ that results from $(T, \mathcal{V})$ by deleting t from T, connecting t' with all vertices that were adjacent to t on T, and setting t' be the root of T' if the deleted t was the root of T remains to be a tree-decomposition of G. Indeed, this pair satisfies (T1) and (T2) as $V_t \subseteq V_{t'}$ holds. As for (T3), let $t_1, t_2 \in T'$ such that $t' \in t_1 T' t_2$. Due to the above construction, $t \in t_1 T t_2$ and hence $V_{t_1} \cap V_{t_2} \subseteq V_t \subseteq V_{t'}$. This means that (T3) holds. We can easily check that for any t_1 on the path from the root to t' on T', $G[V_{\mathrm{sub}(T,t_1)}] = G[V_{\mathrm{sub}(T',t_1)}]$. Therefore, the pair is actually a sub-connected decomposition tree of G of width $\mathrm{tw}(G)$.

Having proposed a procedure to generate a decomposition tree of width $\mathrm{tw}(G)$ by deleting a node of T whose bag is contained in the bag of another, now we can apply this procedure necessary many times to remove all such nodes, and obtain an expected sub-connected decomposition tree of G of width $\mathrm{tw}(G)$. □

Let us investigate some properties of the tree decomposition $(T, \mathcal{V})$ obtained in Lemma 6. Let t_1, t_2 be adjacent bags in T, and T_1, T_2 be two subtrees of T that are obtained by deleting the edge $t_1 t_2$ from T, and without loss of generality, we assume that T_1 is the one including t_1 and T_2 is the other. Lemma 4 says that V_{t_1} contains a vertex in $V_{t_2} \subseteq V_{T_2}$. At the same time, V_{t_1} also contains a vertex that is not in V_{T_2} (since $t_2 \in t_1 T t_3$ for any $t_3 \in T_2$, $v \in V_{t_1} \cap V_{t_3}$ means $v \in V_{t_1} \cap V_{t_2}$ due to (T3)). Thus, V_{t_1} and V_{T_2} are not disjoint but incomparable. Thus, we have

(N2) $|V_t| \geq 2$ for any $t \in T$.

(N3) $|V(T)| \leq |V(G)| - 1$.

For **(N3)**, note that even decomposition tree can be built, starting from one bag, by appending a bag one by one (details will be given in Section 4.4). Imagine that we are building a decomposition tree given by Lemma 6 in this manner. When a bag is added to a decomposition tree $(T, \mathcal{V})$ thus built so far, it must contain a vertex that is not contained in V_T. This means that the bag addition increases the size of underlying graph at least by 1. Thus, the size of T cannot exceed that of G (-1 in the inequality of **(N3)** is because of the lower bound 2 on the bag size **(N2)**).

The upperbound given in **(N3)** cannot be improved anymore. Indeed, if G is a tree, then its tree-width is 1 so that any of its tree decomposition $(T, \mathcal{V})$ of width 1 must satisfy $|V(T)| \geq |V(G)| - 1$ due to (T2).

Theorem 7. *Any (connected) graph $G \in \mathcal{G}$ admits a decomposition tree of width* $\mathrm{tw}(G)$ *that is sub-connected and satisfies* **(N1)**-**(N3)**.

Theorem 8. *For a degree bound Δ, any (connected) graph $G \in \mathcal{G}(\Delta)$ admits a decomposition tree of width* $\mathrm{tw}(G)$ *that is sub-connected, satisfies* **(N1)**-**(N3)**, *and is of degree at most $(w+1)\Delta$.*

Proof. It suffices to prove that when the maximum degree of a given G is Δ, then the decomposition tree given in Theorem 7 is of degree at most $(w+1)\Delta$. Let us denote this decomposition tree by T.

Let us consider adjacent vertices $t_1, t_2 \in T$. Due to Lemma 4, $V_{t_1} \cap V_{t_2}$ is not empty. **(N1)** implies that there is a vertex $v \in V_{t_2} \setminus V_{t_1}$, and (T3) strengthens this as $v \in V_{t_2} \setminus V_{T_1}$. Combining these with the above-mentioned separation property deduces the existence of vertices v_1, v_2 such that $v_1 \in V_{t_2} \cap V_{t_1}$, $v_2 \in V_{t_2} \setminus V_{T_1}$, and v_1v_2 is an edge of the underlying graph G. Consider another bag t_3 that is adjacent to t_1. A similar argumentation as above gives a vertex v_3 that is an analog of t_2 for this bag. We claim that v_3 must be different from v_2; otherwise $v_2(= v_3) \in V_{t_2} \cap V_{t_3}$, and hence, would be in V_{t_1} due to (T3), a contradiction. This claim means that each bag adjacent to t_1 consumes *exclusively* at least one edge of a vertex in V_{t_1}. Consequently, the number of bags adjacent to t_1 is at most $\sum_{t \in V_{t_1}} d(t)$, which is at most $(w+1)\Delta$. □

4.4 Bag addition to decomposition tree

Our algorithm regards the decomposition tree given in Theorem 8 as a representative. Thus, starting from a decomposition tree of one bag, it keeps adding a new bag of size at most $w+1$ (and at least 2 due to **(N2)**) to a decomposition tree so as to generate a representative decomposition tree, whose existence was guaranteed by Theorem 8. As suggested in Section 4.3, a bag V_{t_1} thus added to a bag V_{t_2} of an intermediate decomposition tree $(T, \mathcal{V})$ should satisfy the following two conditions:

1. $V_{t_1} \cap V_{t_2} \neq \emptyset$, that is, $V_{t_1} \cap V(G(T, \mathcal{V})) \neq \emptyset$;
2. $V_{t_1} \setminus V(G(T, \mathcal{V})) \neq \emptyset$.

The number of possible bags is at most $\sum_{i=2}^{w+1} \big(|\Sigma|^i \times (\Delta+1)^{i(i-1)}\big)$; inside the parentheses are the product of the number of ways to label i vertices in a bag and the number of ways to connect two distinct of them by edges, where $2 \leq i \leq w+1$. Anyway, this is constant. There are multiple ways to "overlap" the chosen bag with its parent bag so as to satisfy the above two conditions, but the number is also bounded by a constant. Furthermore, recall that none of the bags of the representative decomposition tree given in Theorem 8 has more than $(w+1)\Delta$ children bags. One problem to be solved yet is of how high the frontier of decomposition trees has to be. This problem will be addressed in the next section.

There is one thing to note. That is about the sub-connectedness. The sub-connectedness of decomposition trees generated in the above method has to be verified; otherwise, the underlying graph of some resulting decomposition tree is not connected, and hence, not even in $\mathcal{G}$. This verification is done recursively. Any two vertices in a leaf-bag of a decomposition tree must be connected. Any two vertices of its non-leaf bag V_t must be connected on the assumption that, for each of its child bags V_{t_1}, any vertices in $V_t \cap V_{t_1}$ are connected. See Figure 4 as an example. The leftmost decomposition tree, which consists of one bag

$\{1, 3, 5\}$, is not sub-connected, and our algorithm has to append a child to it. Even after adding a new bag $\{1, 2, 3\}$, still the resulting decomposition tree is not sub-connected at its root. By sharing the vertices $1, 3$, the root bag (parent) can delegate the responsibility to connect these to the child bag (and to other children to be added from now). Here an essential issue occurs; our algorithm does not append any vertex to the bag $\{1, 2, 3\}$ because otherwise it would not be able to append a child to the root that takes a responsibility to connect the currently-isolated vertex 5 with 1 or 3. Adding $\{3, 4, 5\}$ to the root finally results in a sub-connected decomposition tree. What we wanted to insist on is that sub-connectedness of a decomposition tree can be guaranteed during its growth process locally by each of its vertices checking which of its vertices are shared with each of its child bag.

4.5 An algorithm for Solvability$(\boldsymbol{S}_K, \mathcal{TW}(2, \Delta))$ that runs in pseudo-polynomial time

We are now ready to present our algorithm. It is a generalization of A-F algorithm modified so as to run the dynamic programming on decomposition trees given in Theorem 8 instead of on trees. Our algorithm is, however, not guaranteed to work in polynomial time in n.

In this section, we prove that any graph G of width at most 2 admits a decomposition tree of width $\mathrm{tw}(G)$ such that, for any $v \in G$, $|T_v| \leq \frac{3}{2}\Delta(G)$ (Lemma 10). Since our algorithm assumes the degree bound Δ, this means that it suffices to consider decomposition trees on which the size of T_v is bounded by $\frac{3}{2}\Delta$. Recall that $v_1v_2 \in E(G)$ implies $T_{v_1} \cap T_{v_2} \neq \emptyset$. Thus, our algorithm can set the height of frontier to $\frac{3}{2}\Delta(K + 1)$. Let us prove Lemma 10, which is almost a corollary of the next lemma.

Lemma 9. *Any series-parallel graph G admits a tree-decomposition $(T, \mathcal{V})$ of width at most 2 such that $|T_v| \leq d(v) + 1$ for any $v \in G$.*

Proof. Our proof is constructive. Given an atomic series-parallel graph whose terminals correspond to the vertices v, v' of G, a tree of one bag $\{v, v'\}$ (root) is clearly its tree-decomposition of width at most 2. For each edge of G, we have a corresponding atomic series-parallel graph. Thus, we can say that the number of bags containing $v \in G$, which we denote by $\beta(v)$ in this proof, is equal to $d(v)$ at the beginning.

Now we propose a method to build up a tree-decomposition $(T, \mathcal{V})$ of G from these atoms without increasing $\beta(v)$ by no more than 1 for every $v \in G$. For induction, we assume that the root of any intermediate tree-decomposition is the bag that consists of the terminals of underlying graph. This assumption holds for the atoms.

Parallel operation is handled first. Let G_1, G_2 be series-parallel graphs whose terminals correspond to the same two nodes $v_a, v_b \in G$. Let $(T_1, \mathcal{V}_1), (T_2, \mathcal{V}_2)$ be their respective tree-decompositions. Due to the inductive hypothesis, the roots of both decompositions are $\{v_a, v_b\}$. Hence, when parallel operation is

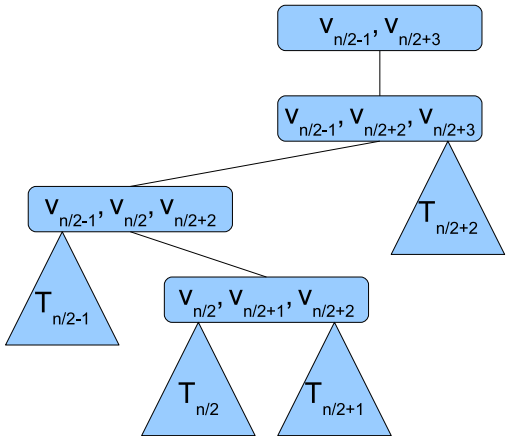

Figure 6: A snapshot of a construction of a tree-decomposition of a graph that is obtained by the series operation of graphs $G_1, \ldots, G_{n-1}$ with respective tree-decompositions $T_1, \ldots, T_{n-1}$. For clarity, the floor function is omitted here; $n/2$ in this figure should be $\lfloor n/2 \rfloor$.

applied to G_1 and G_2, we simply merge the root of T_1 with that of T_2. The pair of the tree thus obtained and $\mathcal{V}_1 \cup \mathcal{V}_2$ becomes a tree-decomposition of the resulting graph. This operation decreases $\beta(v_a)$ and $\beta(v_b)$ by 1. In contrast, for any internal vertex v of G_1 or G_2, $\beta(v)$ does not change, and this is true also for series operation discussed below.

Series operation needs a more involved analysis. What we will actually consider is a succession of series operations of maximal length. That is, given $n-1$ series-parallel graphs G_i whose terminals correspond to $v_i, v_{i+1} \in G$ and their tree-decompositions $(T_i, \mathcal{V}_i)$ for $1 \leq i \leq n-1$, the operation merges the v_{j+1} of G_j with the v_{j+1} of G_{j+1} for $1 \leq j \leq n-2$ to construct a series-parallel graph G'. The terminals of G' will be v_1, v_n. By maximal length, we mean that either $G' = G$ or it is the parallel operation that G' will get involved in next.

The construction of a tree-decomposition of G' is as follows. First the root $\{v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor+1}\}$ of $T_{\lfloor n/2 \rfloor}$ is merged with the root $\{v_{\lfloor n/2 \rfloor+1}, v_{\lfloor n/2 \rfloor+2}\}$ of $T_{\lfloor n/2 \rfloor+1}$ into a bag $\{v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor+1}, v_{\lfloor n/2 \rfloor+2}\}$, and to this bag, $\{v_{\lfloor n/2 \rfloor}, v_{\lfloor n/2 \rfloor+2}\}$ is appended as the root of the resulting decomposition tree. To this resulting tree, $T_{\lfloor n/2 \rfloor-1}$ is appended in the same manner, and then, $T_{\lfloor n/2 \rfloor+2}$ is appended and so on until T_1, T_{n-1} are thus appended (see Figure 6).

This operation may increase $\beta(v_1), \beta(v_2), \ldots, \beta(v_n)$, but at most by 1. Since $v_2, \ldots, v_{n-1}$ become internal in this operation, this is the last increase possible. As for v_1, v_n, the increase might seem to be 2, but this is due to the addition of root bag $\{v_1, v_n\}$ for the next parallel operation, which will decrease the number for v_1, v_n by 1 as mentioned above. If no parallel operation follows, we can simply remove this root bag. Thus, in any case, the increase amounts to at

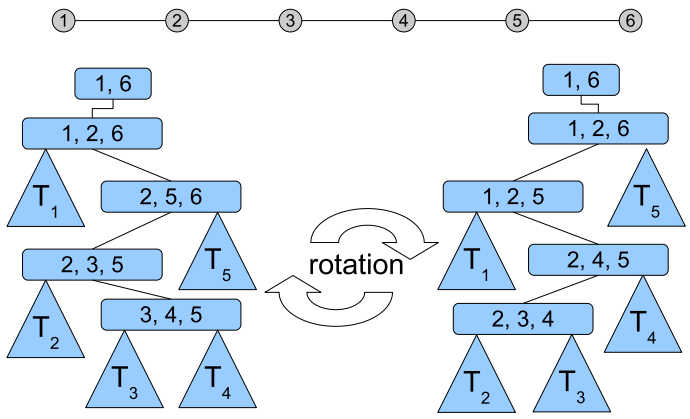

Figure 7: Two decomposition trees of a graph with 6 vertices $v_1, v_2, \ldots, v_6$ such that v_i is connected with v_{i+1} for $1 \leq i < 6$ (shown at the top). In the left one, the terminal 1 is contained only in 2 bags, while in the right one, the terminal 6 is contained only in 2 bags. Tree rotation converts one into the other.

most 1 so that we can say that $\beta(v_1), \beta(v_n)$ can increase by at most 1. In fact, only one of $\beta(v_1)$ and $\beta(v_n)$ can increase and the other remains. Note that v_1 or v_n will become internal in the coming series operation (not current one, but next one). Thus, it is desirable for the current series operation to increase the number of bags containing the terminal that becomes internal in this way. It was $\beta(v_n)$ that increased in the above-mentioned construction of decomposition tree. If $\beta(v_1)$ should be increased, then we should first merge rather $T_{\lfloor n/2 \rfloor - 1}$ and $T_{\lfloor n/2 \rfloor}$, followed by the merging of $T_{\lfloor n/2 \rfloor + 1}$, then merge $T_{\lfloor n/2 \rfloor - 2}$, and so on (this can be easily understood in the context of tree rotation, see Figure 7).

According to the above construction, the number of bags containing a vertex increases only when it shifts from a terminal to an internal, and the increase is at most 1. Recall that originally $\beta(v) = d(v)$ for each $v \in G$. Let us conclude this proof by checking that the above construction does not destroy the requirements (T1)-(T3) for tree-decomposition. Since our construction is based on merging of bags and at the beginning of the construction we have a bag $\{v, v'\}$ for every edge $vv' \in G$, it is obvious that the resulting decomposition tree has a bag that contains both v and v'. Thus, (T2) holds. This, in turn, implies (T1) because G is a connected graph. The verification for (T3) is done by induction on the series and parallel operations. For decomposition trees of size 1, (T3) is obviously true, and hence, (T3) is true at the beginning of our construction. Afterwards, it suffices to check that merging of the roots of two intermediate trees results in another intermediate tree T that satisfies (T3) assuming that the given two intermediate trees satisfy (T3). Proving that T satisfies (T3) is equivalent to proving that for any vertex v, T_v is a connected component of T. Recall that the root of intermediate trees is of size 2. Let us denote these

trees by T_1 and T_2 with the respective roots $\{u_1, u_2\}$ and $\{v_1, v_2\}$. By definition of series and parallel operations, if a vertex appears in both T_1 and T_2, then it is one of u_1, u_2, v_1, v_2. Thus the assumption that T_1 and T_2 satisfy (T3) makes it sufficient to prove that $T_{u_1}, T_{u_2}, T_{v_1}, T_{v_2}$ are connected components of T. Due to the assumption, T_{1u_1} and T_{2u_2} are connected, and furthermore, contain the respective roots of T_1 and T_2 unless they are empty. Thus, T_{u_1} is a connected component of T. This is true for u_2, v_1, v_2. Consequently, T also satisfies (T3). □

We have mentioned in Section 4.2 that any graph of tree-width at most 2 can be written as a tree of biconnected components that are series-parallel. Given a graph G of tree-width at most 2, we construct its decomposition tree of width $\mathrm{tw}(G)$ in the following manner. To each of its biconnected component, we first apply Lemma 9 and obtain its decomposition tree of width at most $\mathrm{tw}(G)$. Consider two biconnected components that share a vertex $v \in G$ and their decomposition trees $(T_1, \mathcal{V}_1), (T_2, \mathcal{V}_2)$ thus obtained. By definition, there must exist $t_1 \in T_1$ and $t_2 \in T_2$ such that $v \in V_{t_1}$ and $v \in V_{t_2}$. A new edge t_1t_2 connects T_1 and T_2 into a tree T_3. Then $(T_3, \mathcal{V}_1 \cup \mathcal{V}_2)$ is a decomposition tree of the subgraph of G that consists of these two biconnected components. This decomposition tree is of width at most $\mathrm{tw}(G)$. In this way, we connect all decomposition trees for the biconnected components of G into one decomposition tree $(T, \mathcal{V})$.

We claim that this decomposition tree satisfies that for any $v \in G$, $|T_v| \leq \frac{3}{2}d(v)$. Assume that v is contained in k biconnected components of G for some $k \geq 1$, and that each of the first j components consists of at least 3 vertices of G; the others consist of 2 vertices. For $1 \leq i \leq k$, let $(T_i, \mathcal{V}_i)$ be the decomposition tree of the i-th biconnected component obtained by Lemma 9. Let $d_1, d_2, \ldots, d_k$ be the number of edges adjacent to v that these components contain. Since distinct biconnected components cannot share any edge of G, $\sum_{i=1}^{k} d_i = d(v)$. Note that if the i-th biconnected component consists of at least 3 vertices, then $d_i \geq 2$. With $\sum_{i=1}^{j} d_i \leq d(v)$, this implies that $j \leq \frac{1}{2}d(v)$. Otherwise the component admits a trivial decomposition tree (only one node). According to the above-mentioned construction of $(T, \mathcal{V})$,

$$\begin{aligned} |T_v| = \sum_{i=1}^{k} |T_{iv}| &\leq \sum_{i=1}^{j} (d_i + 1) + k - j \\ &\leq \sum_{i=1}^{j} d_i + k - j + j \\ &\leq d(v) + j \leq \frac{3}{2} d(v), \end{aligned}$$

where we employ $k - j \leq \sum_{i=j+1}^{k} d_i = d(v) - \sum_{i=1}^{j} d_i$. Now we have verified the next lemma.

Lemma 10. *Any graph G of tree-width at most 2 admits a decomposition tree $(T, \mathcal{V})$ of width* $\mathrm{tw}(G)$ *such that $|T_v| \leq \frac{3}{2}d(v)$ for any $v \in G$.*

Theorem 11. *Let Δ be a degree bound. For the class $\mathcal{TW}(2, \Delta)$, our algorithm runs in polynomial time in n.*

The height of T_v is closely related to the efficiency of our algorithm. Any graph of tree-width 1 admits a decomposition tree T such that the height of T_v is at most 2. Is it also the case that any graph of tree-width 2 admits a decomposition tree such that the height is bounded by a constant that does not depend on Δ? A negative answer is to be found in Figure 8. Let $G = (\{v_0, v_1, \ldots, v_n, v'_1, \ldots, v'_n\}, E)$ be the graph in this figure, where the vertex v_i is specified by its subscript for the clarity. This graph, G, is a series-parallel graph, and hence, admits a tree-decomposition $(T, \mathcal{V})$ of width 2. It is known that the three vertices of any 3-clique (triangle) in G must be in one bag [5]. For example, the vertices v_0, v_1, v_2 form a triangle so that $(T, \mathcal{V})$ must have a bag $\{v_0, v_1, v_2\}$. In this way, we can figure out that $\{v_0, v_1, v_2\}$, $\{v_0, v_2, v_3\}$, $\{v_0, v_3, v_4\}$, $\ldots$, $\{v_0, v_{n-1}, v_n\}$, and their counterparts for $v'_1, v'_2, \ldots, v'_n$ as well as $\{v_0, v_n, v'_n\}$ are the bags that any tree-decomposition of G of width 2 must contain. Note that the vertices of T to which these bags are assigned must induce the connected subgraph of T. The only way to connect them without destroying (T3) is by connecting them in a line as illustrated at the bottom of this figure. Then, T_{v_0} is the decomposition tree itself because all of the bags of the tree contains the vertex v_0. Hence, even if we choose the bag $\{v_0, v_n, v'_n\}$ as a root, T_{v_0} is still of height $n - 1$, and this is the best choice of root to make T_{v_0} the shallowest. Note that the degree of the vertex v_0 is $2n$.

5 Conclusion

In this paper, we have proposed a comprehensive framework of inferring graphs in a given object space from a given walk history or system of walk histories. This integrated results known in formal language theory and in chemoinformatics with new results, and as a consequence, we obtained a profound understanding of structure elucidation problem. For the class of graphs of tree-width at most 2, we proposed a pseudo-polynomial time algorithm. Based on this, we should design an efficient enumeration algorithm as done in [7]. Such algorithms for larger tree-width are also open.

6 Appendices

6.1 A proof of Theorem 3

Let us propose two results first, which are useful in constructing a pseudo polynomial time transformation from 3-PARTITION to $\textsc{Solvability}(\boldsymbol{S}_2, \mathcal{PLG})$. The graphs considered from now on are assumed to be *undirected*. A vertex v_1

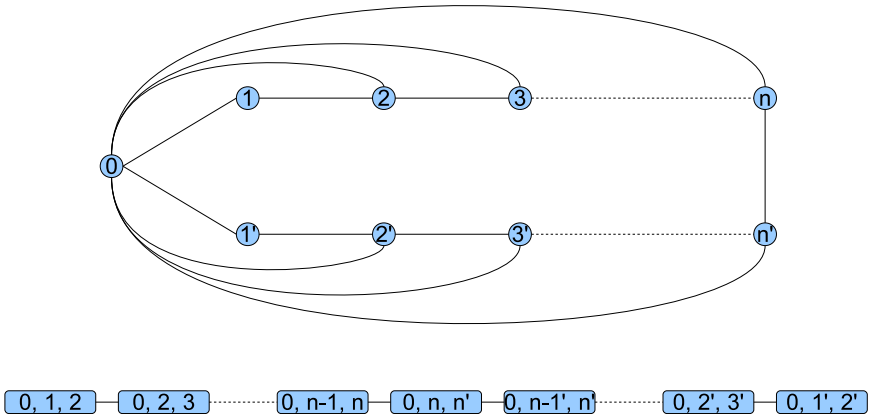

Figure 8: A graph of tree-width 2 such that for any of its tree-decomposition, the height of subgraph induced by T_{v_0} for the vertex v_0 is at least $n-1$. The bottom illustrates the subtree that any tree-decomposition of width 2 needs to contain.

of a graph is *singly connected* with another vertex v_2 if there exists exactly one edge between them.

Lemma 12. *For an undirected graph G, if G contains exactly one a-vertex and $|G|_{ab} = |G|_{aba} = n$ for some n, then there exist n b's that are singly connected with the a-vertex.*

Proof. Let v be the a-vertex of G. An ab-walk contributes to $|G|_{aba}$ by 1 (starting from the a-vertex, we arrive at the b-vertex via the edge on this walk and return back via the same edge). Hence, $|G|_{ab} \leq |G|_{aba}$ holds.

Suppose that a b-vertex v_1 is connected with v by two edges e_1, e_2. Then apart from the aba-walks explained above, now we have extra two aba-walks, that is, $ve_1v_1e_2v$ and $ve_2v_1e_2v$. Then, G would contain at least $n+2$ aba-walks, a contradiction.

Since G contains n ab-walks, exactly n b-vertices are singly connected with the a-vertex. □

Lemma 13. *For an undirected graph G, if G contains n a-vertices and n b-vertices and $|G|_{ab} = |G|_{aba} = |G|_{bab} = n$ for some n, then there exist n pairwise distinct pairs of an a-vertex and b-vertex that are singly connected.*

Our proof of the following theorem will borrow several basic terminologies from topology. Let G be a planar multigraph, that is, G can be embedded onto the plane $\mathbb{R}^2$. The regions of $\mathbb{R}^2 \setminus G$ are called the *faces* of G. Since we can lay G inside some sufficiently large disc D, there exists exactly one among its faces that cannot be thus bounded, that is, the face that contains $\mathbb{R}^2 \setminus D$. This face is called the *outer face* of G, and the others are called its *inner faces*.

Now, we are ready for proving Theorem 3.

Proof. The basic idea is from [1]: a pseudo polynomial time transformation from 3-PARTITION, which is defined as: given a set X that consists of $3m$ elements $x_1, \ldots, x_{3m}$ along with their integer weights $w(x_i)$ and a positive integer B such that $B/4 < w(x_i) < B/2$ for $1 \leq i \leq 3m$, find a partition of X into m (disjoint) sets $A_1, \ldots, A_m$ of cardinality 3 such that $A_j = \{x_{j,1}, x_{j,2}, x_{j,3}\}$ and $w(x_{j,1}) + w(x_{j,2}) + w(x_{j,3}) = B$ for $1 \leq j \leq m$, where $x_{j,1}, x_{j,2}, x_{j,3} \in X$.

Let $\Sigma = X \cup \{a_1, \ldots, a_m\} \cup \{a, b, c, d, f_1, f_2\}$. From a given instance of 3-PARTITION, we construct a feature vector $\boldsymbol{v}$ of level 2 specified as follows; we write $x_i = 1$ to indicate that the x_i coordinate of $\boldsymbol{v}$ has value 1. For any u, if the value of u coordinate of $\boldsymbol{v}$ is not mentioned below, then it is 0, that is, in the target graph, no u-walk is found. For $1 \leq i \leq 3m$ and $1 \leq h \leq m$,

VERTICES $x_i = 1$, $a = Bm$, $b = 3m$, $c = 3m + 1$, $d = 1$, $f_1 = f_2 = 3m$, and $a_h = 1$;

WALKS-C(enter) $da_h = 1$, $a_h b = a_h b a_h = 3$, and $a_1 a_2 = a_2 a_3 = \cdots = a_{m-1} a_m = 1$;

WALKS-B(lock) for $s \in \{1, 2\}$, $bf_s = bf_s b = f_s bf_s = 3m$, $x_i f_s = 1$, $ba = bab = Bm$, $x_i a = x_i a x_i = w(x_i)$;

WALKS-BC $x_i d = 1$, $f_1 c f_2 = 3m - 1$, $a_h c = a_h c a_h = 3$, $f_1 c a_1 = 3$, $f_2 c a_1 = 2$, $f_1 c a_\ell = 3$, $f_2 c a_\ell = 3$ for $1 < \ell < m$, $f_1 c a_m = 3$, $f_2 c a_m = 4$, and $a_h b a = B$;

WALKS-I(nhibited) for any $1 \leq j, k \leq m$ with $j \neq k$, $x_j f_1 x_k = x_j f_2 x_k = a_j b a_k = a_j c a_k = 0$.

For example, $x_i = 1$ in VERTICES means that a target graph must contain exactly 1 x_i-vertex.

Let us give a topological characterization of graphs $G \in \mathcal{PLG}$ that satisfy $|G|_{\boldsymbol{S}_2} = \boldsymbol{v}$. Indeed, we shall see that $\boldsymbol{v}$ uniquely determines a structure that consists of the *center graph* (an m-star with the center d-vertex) and the a_h-vertex ($1 \leq h \leq m$), to each of which 3 b-vertices are singly connected) and $3m$ rhombuses bounded by the cycle $bf_1 x_i f_2 b$ (x_i-*rhombus*), within which exactly $w(x_i)$ a-vertices are forced to be fenced and single connected with the b-vertex on the rhombus (see Figure 9). Once confirmed, this structure and its uniqueness enable us to conclude that the given instance of 3-PARTITION has a solution if and only if there exists a planar graph whose feature vector of level 2 is $\boldsymbol{v}$; this is because $a_h b a = B$ must be satisfied for $1 \leq h \leq m$. Note that our construction of the system of inequalities is a pseudo polynomial time transformation.

First of all, VERTICES, WALKS-C, and WALKS-I determine the center graph. Due to Lemma 12, $a_h b = a_h b a_h = 3$ force exactly 3 b-vertices be singly connected with each a_h-vertex and $a_j b a_k = 0$ in WALKS-I inhibits a b-vertex from being connected with more than one of $a_1, \ldots, a_m$-vertices. For $1 \leq \ell < m$, the a_ℓ-vertex has to be connected with the $a_{\ell+1}$-vertex in order to satisfy $a_\ell a_{\ell+1} = 1$.

We shift our focus onto the x_i-rhombus and $w(x_i)$ a-vertices. As being done above but using Lemma 13, one can easily see that exactly one of $3m$ f_1 (f_2)-vertices must be singly connected to each of the $3m$ b-vertices of the center graph in a one-to-one manner. To these f_1-vertices, distinct x_i-vertex is to be singly connected as we need exactly one $x_i f_1$-walk and $x_j f_1 x_k$-walk is inhibited whenever $j \neq k$. This fact allows us to index the f_1-vertex and b-vertex on the walk from the x_i-vertex to the d-vertex by the subscript i as $f_{1,i}$ and b_i, and the f_2-vertex that is connected to the b_i-vertex is thus indexed as $f_{2,i}$, but this indexing is only for the ease of explanation. In the following, we denote the three of $x_1, \ldots, x_{3m}$-vertices that have been thus connected with the a_1-vertex by $x_{1,1}$, $x_{1,2}$, and $x_{1,3}$ for convenience sake (see Figure 9), but note that we do not know which of $x_1, \ldots, x_{3m}$ is $x_{1,1}$ or we should not. The extended center graph built so far is still a tree, and hence, has only one face. Now we draw edges from each of these x_i-vertices to the d-vertex, but due to the $a_1 a_2 \cdots a_m$-walk, these edges cannot help but go through between the a_1-vertex and a_m-vertex as shown in Figure 9. These edges separate the face into $3m + 1$ faces, that is, the face bounded by the $da_1 b f_1 x_{1,1}$-walk, one bounded by the $dx_{1,1} f_1 b a_1 b f_1 x_{1,2} d$-walk, and so on. Since the $x_j f_2 x_k$-walk is inhibited whenever $j \neq k$, each x_i-vertex must be singly connected with distinct f_2-vertex. The lines from x_i to d now force the x_i-vertex to be thus connected with the $f_{2,i}$-vertex. As a result, the x_i-rhombus has been formed.

Now we will fence in $w(x_i)$ a-vertices in the x_i-rhombus. To this end, we connect the $3m$ f_1-vertices and f_2-vertices via $3m - 1$ $f_1 c f_2$-walks. The readers should be now familiar enough with the technique based on Lemma 12 and WALKS-I to check that exactly 3 c-vertices are singly connected with the a_h-vertex, and these be distinct. This means that the c-vertex on any of these $f_1 c f_2$-walks must be connected with the a_h-vertex for some h, and hence, none of these walks can share their f_1-vertex and f_2-vertex. It is left to the reader to check that the way illustrated in Figure 9 is the only way to draw $3m - 1$ $f_1 c f_2$-walks so as to satisfy all of these requirements and $f_1 c a_1 = 3$, $f_2 c a_1 = 2$, $f_1 c a_\ell = 3$, $f_2 c a_\ell = 3$ for $1 < \ell < m$, $f_1 c a_m = 3$, $f_2 c a_m = 4$. These newly-added structures prevent an a-vertex from being connected both with a b-vertex and with x_i-vertex unless it is placed in the x_i-rhombus. Check that each a-vertex must be singly connected with exactly one b-vertex, and $w(x_i)$ a-vertices must be singly connected with the x_i-vertex. Thus, the x_i-rhombus must contain exactly $w(x_i)$ a-vertices and they have to be singly connected with the b_i-vertex. □

7 Acknowledgements

We wish to express our gratitude for the anonymous referees for their carefully and thoroughly reviewing the earlier version of this manuscript and giving valuable comments and suggestions on it. Shinnosuke Seki expresses his sincere gratitude to Professor Mark Daley, Professor Oscar. H. Ibarra, Professor Helmut Jürgensen, Professor Lila Kari, and Professor Arto Salomaa for the creative discussions with them on the research topic in this paper.

This research was carried out with the financial support of the JSPS Postdoctoral Fellowship P10827 to Szilárd Zsolt Fazekas, of the Funding Program for Next Generation World-Leading Researchers (NEXT program) to Yasushi Okuno, and of the Kyoto University Start-up Grant-in-Aid for Young Scientists, No. 021530, to Shinnosuke Seki. Works by Shinnosuke Seki were also financially supported by Department of Information and Computer Science, Aalto University.

References

[1] Akutsu T, Fukagawa D (2005) Inferring a graph from path frequency. In: Aposolico A., Crochemore M., Park K. (eds.), CPM 2005, Lecture Notes in Computer Science, vol 3537, Springer, pp. 371–382.

[2] Bakir G H, Weston J, Schölkopf B (2004) Learning to find pre-images. In: Advances in Neural Information Processing Systems, pp. 449–456.

[3] Bakir G H, Zien A, Tsuda K (2004) Learning to find graph pre-images. In: Proceedings of the 26th DAGM Symposium, Lecture Notes in Computer Science, vol 3175, Springer, pp. 253–261.

[4] Bodlaender H (1998) A partial k-arboretum of graphs with bounded treewidth. Theoret. Comput. Sci. 209 (1-2): 1–45.

[5] Diestel R (2010) Graph Theory, 4th Edition. Springer.

[6] Fraigniaud P, Nisse N (2006) Connected treewidth and connected graph searching. In: LATIN 2006. Lecture Notes in Computer Science, vol 3887, Springer, pp. 479–490.

[7] Fujiwara H et al (2008) Enumerating treelike chemical graphs with given path frequency. Journal of Chemical Information and Modeling 48:1345–1357.

[8] Garey M R, Johnson D S (1979) Computers and Intractability. A Guide to the Theory of NP-Completeness. W. H. Freeman and Co.

[9] Goto S et al (2002) LIGAND: Database of chemical compounds and reactions in biological pathways. Nucleic Acids Research 30:402–404.

[10] Ibarra OH (1978) Reversal-bounded multicounter machines and their decision problems. Journal of the ACM 25:116–133.

[11] Leslie C, Eskin E, Noble WS (2002) The spectrum kernel: A string kernel for SVM protein classification. In: Proceedings of the 7th Pacific Symposium on Biocomputing. pp. 564–575.

[12] Mateescu A, Salomaa A, Yu S (2004) Subword histories and Parikh matrices. J. Comput. Syst. Sci. 68:1–21.

[13] Matiyasevich Y (1970) Solution of the tenth problem of Hilbert. Matematikai Lapok 21:83–87.

[14] Matiyasevich Y (1993) Hilbert's Tenth Problem. MIT Press.

[15] Nagamochi H (2009) A detachment algorithm for inferring a graph from path frequency. Algorithmica 53:207–224.

[16] Parikh RJ (1966) On context-free languages. Journal of the Association for Computing Machinery 13:570–581.

[17] Robertson N, Seymour PD (1986) Graph minors. ii. algorithmic aspects of tree-width. Journal of Algorithms 7:309–322.

[18] Rozenberg G, Salomaa A (eds) (1997) Handbook of Formal Languages, vol 1. Springer.

[19] Seki S (2011) Absoluteness of subword inequality is undecidable. Theor Comput Sci 418:116-120.

[20] Shannon CS, Weaver W (1949) The Mathematical Theory of Communication. The University of Illinois Press.

[21] Yamaguchi A, Aoki KF, Mamitsuka H (2003) Graph complexity of chemical compounds in biological pathways. Genome Informatics 14:376–377.

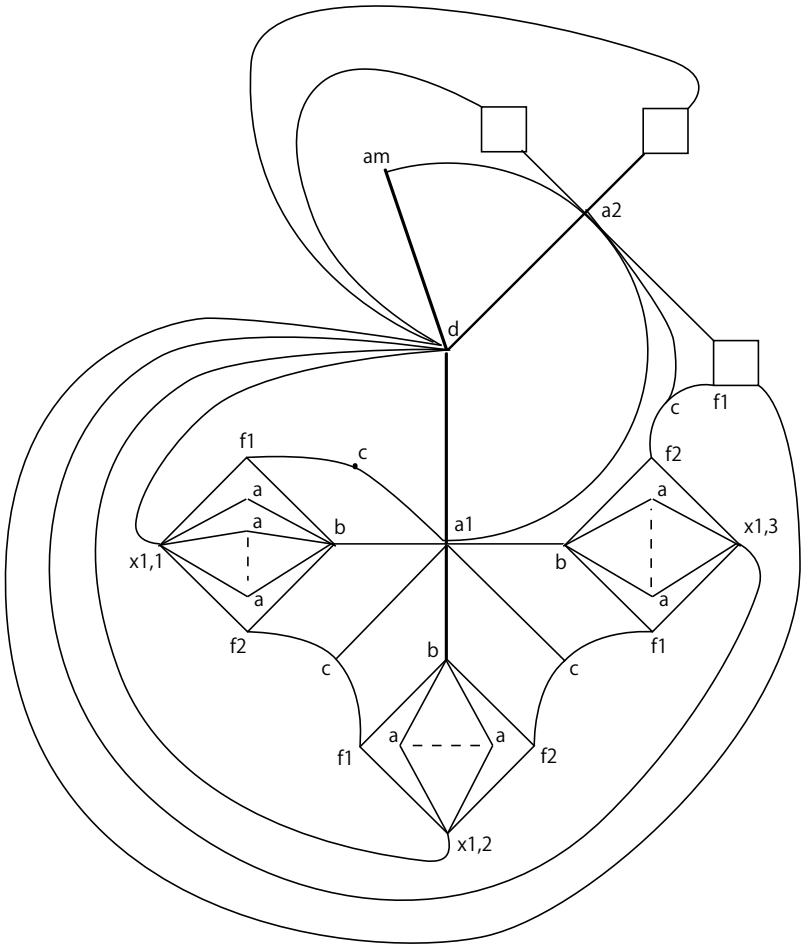

Figure 9: Reduction from 3-PARTITION to Solvability($\boldsymbol{S}_2, \mathcal{PLG}$).